Implementation of a T6963C LCD Device Driver in Linux 2.6

Jacob Farkas
CPE 454

March 11, 2005

Contents

1 Introduction **3**

2 Theory of Operation **3**

3 Hardware Setup **4**

4 Implementation **4**

5 Problems Encountered **7**

6 Conclusion **8**

1 Introduction

I purchased an LCD screen from eBay that can display 30x6 characters or 240x48 pixels of graphics. The display is controlled by a Toshiba T6963C Dot Matrix Controller chip. A Linux driver exists for the T6963C already, written by Alexander Frink[3], but the driver is written for Linux 2.2 and I wanted the experience of writing a device driver for Linux 2.6.

The goal for this project is to create a Linux 2.6 character driver that interfaces with the T6963C and allows read and write access to the screen. My final goal is to write some scripts to capture wave information from the NOAA and CDIP websites and display surf reports on the LCD screen. I would also like to expand the character driver into a full-featured console driver similar to the 2.4 driver written by Alexander Frink.

2 Theory of Operation

LCD screens with parallel interfaces can be accessed by connecting pins on a PC's parallel port to pins on the LCD display controller. Under Linux, the parallel port's pins are memory mapped as an I/O port, usually starting at address 0x378. Using the `outb()` and `inb()` functions, one can read from and write to the parallel port. The T6963 datasheet[2] explains the T6963 operations, and Steve Lawther has written an excellent guide[1] on writing software for the T6963.

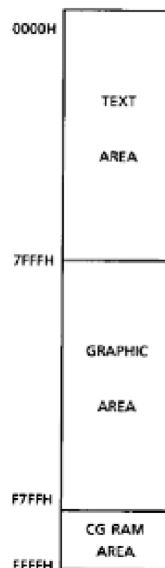

Figure 1: T6963 memory allocation

The T6963 has 64kB of memory available. This memory is split up into a text area, a graphics area, and a character generator area. These memory areas are configurable by the user. Since my display is 240x40 (30 characters by 5 rows), I only need 150 bytes of space for text memory and 1200 bytes for graphics memory. This leaves a large amount of memory available. This extra memory can be used for on-chip framebuffering- the next screen of data could be written to empty memory space and the graphics area could be changed to this memory space when the write is complete.

The display is only two colors; each pixel can only be on or off. I discovered that the latency of the display is high enough that grayscale could be faked by updating certain pixels only intermittently. This could be done by generating multiple framebuffers. The darkest pixels would be in all of the framebuffers while the lightest pixels would only be in one framebuffer. The framebuffers would be cycled and the latency of the pixels would create a psuedo greyscale display.

3 Hardware Setup

The LCD display is powered by a +5VDC supply, so I needed to obtain an AC/DC convertor that output +5VDC. The website allelectronics.com had a great deal on Ericsson switching power supplies, built originally as cell phone chargers, for $4.50. Since I will be connecting the display through the PC's parallel port, I purchased a few 25 pin sub mini D connectors and cases from the same site.

Connecting to the LCD's input pins was a bit more difficult. I ordered some sample cables from samtec.com but couldn't find any that matched the pinout. I discovered that the pin spacing is exactly the same as an IDE cable, so I connected an IDE cable to the display and soldered wires to the IDE cable and into the parallel port.

4 Implementation

Since I bought the screen off eBay, I couldn't be certain that the hardware actually worked. The most difficult part of writing this driver was setting up and familiarizing myself with the equipment. My first attempt at a device driver was to simply take the device driver code we had been given in class and add functions to command and write to the T6963.

After a long night I managed to get text to appear on the display, confirming that the hardware worked. I used the internal character generator to display the text, but I wanted to use the graphics capabilities of the chip to do all of the display. After a couple more hours I managed to get a bitmap to display on the screen.

(a)

(b)

Figure 2: (a) Bitmap file, (b) Bitmap on LCD screen

My initial code was nothing more than a proof of concept and was very messy. Once I understood the approach to displaying graphics on the screen, I decided to rewrite the driver to clean it up and also with the goal of writing the driver as a console driver, using Frink's driver and the Linux `mdacon` driver as a reference.

My first attempt used an `outb_p()` for all output, which delays for about 1ms. I masked in the bits that i needed for each output in one call, for example `outb_p(0x20 | CTRL_CMD | CTRL_READ | CTRL_CE, CTRL);` to set up the port to read the status. This was an unwieldily way of outputting bits on the port, as I had to keep track of which were active high and active low as mentioned above. After looking at some other LCD drivers it looked like the common format was to write macros that took the contents of the control register and masked in the appropriate bit. For example, `#define READ_H outb_p(inb(CTRL) | CTRL_READ, CTRL)` to set the read bit. These macros are then called in order to set up the port to the state desired. I used this method for my second implementation and added a bit of a speedup by making all the macros a non-delayed `outb()` and adding a `DELAY` macro at the end of every state setup that wrote to a delay port (port 0x80).

The LCD screen provides for reading of the contents of its memory, but I didn't see much use to doing so. I implemented a `read()` function in my driver but due to lack of time it is untested and I didn't run into a reason to read from the display.

I wrote two programs that interfaced with the graphics driver. The first program I wrote displayed a 1-bit bitmap file to the screen. I referenced a web page written by Stefan Hetzl describing the .bmp file format[4] to write a .bmp file decoding function. I quickly realized the limitations of only

having 256x40 pixels of space to work with, so the next feature I worked on was scrolling. Scrolling the image byte by byte was too choppy so I wrote a function that precalculated eight single bit shifts of the data and cycled through those arrays to create single bit scrolling. The single bit scrolling looks very smooth and can run at varying speeds using a `usleep()` call in the update loop.

(a) (b) (c)

Figure 3: (a) Original image, (b) Bitmap sent to LCD (c) Simulated grayscale image displayed on LCD

After noticing the latency of the pixels during a fast scroll I realized that even though the screen is only 1 bit I could simulate grayscale on the display by refreshing the pixels at different rates. I wrote a program that reads 8-bit bitmaps and creates a series of 1-bit bitmaps that correspond to different intensity levels. At 1280 bytes per buffer, I found that there is enough room in the T6963 memory to store multiple buffers directly on the T6963 chip. This saves a great amount of time displaying the image as all that needs to be sent is a command to the T6963 telling it to look at a new region of memory as the graphics base. The image data only needs to be written to the chip once.

The scrolling bitmap program sent the data for the entire contents of the

screen every time it refreshed, and therefore was much more CPU intensive. `top` showed the bitmap program at approximately 45% CPU usage while the grayscale program used so little I couldn't even get it to show up on the list of top processes. Keeping as much data on the chip is very efficient.

5 Problems Encountered

The first problem I encountered was that the display had five control lines but aside from the data lines, I could only control four output lines on the parallel port. This was more than a little frustrating, since there are more than enough available pins on the parallel port connector. The T6963 data sheet says that "After power on, it is necessary to reset. /RESET is kept L between 5 clocks up (oscillation clock)." Luckily, Alexander Frink noticed the same problem and comments on his website: "My display runs fine without ever connecting the Reset pin. This is not clear from the documentation of the controller. Unfortunately the parallel port has only 12 bits which are needed for D0-D7, C/D, /WR, /RD and /CE, so there is no easy possibility to programmatically reset the display. It might be possible with some logic IC combining the output of /RD and /WR, but this is not included in the driver."[3]. I left the /RESET pin tied to the +5VDC line and the display works fine without pulling it low.

The eBay seller of the LCD screen listed what he claimed was the pinout for the screen on the auction page. I connected the wires according to his pinout but I could not get a status response from the LCD screen. I finally sat down and followed each trace on the PCB board from the pinout to the T6963C chip and compared the pins with the T6963C data sheet[2]. The seller had the /RD and /WR lines on the wrong pins. To fix this I changed the /RD and /WR lines in software.

The active low lines also caused a bit of confusion. The values in bits 0,1, and 3 of the control register are flipped before they are put on the parallel port pins. To add to that confusion, the READ and WRITE lines on the T6963C are active low. After I sorted all of this out it ended up being fairly simple; to perform a read, the READ bit should be a 1 in the register. This bit is then flipped before being put on the pin, which is an active state for the READ line.

I learned about the bidirectional port bit in the control register the hard way. This bit controls whether bits are read from or written to the parallel port. The BiDi bit is bit 5 of the control register. A 1 in this bit position allows the flow of data in to the parallel port, a 0 in this bit position allows the flow of data out from the parallel port control registers. Having code that matches the data sheet exactly, yet still won't display anything on the LCD screen because the BiDi bit wasn't set was a trying experience.

6 Conclusion

I had a lot of fun writing the software for the T6963 display. There were a couple of initial hurdles to interfacing with the display, but one I had one display command figured out everything else came together quickly. Writing code and being able to see the results of it on a screen was rewarding and I was especially happy seeing how well the grayscale emulation on the display worked out.

The display can perform bitwise logical operations on the contents of display memory, so it's possible to display both graphics and text on the screen at the same time. In keeping with my original plan of using the display to show current surf information, I would like to make part of the screen display wave models while the rest of the screen holds text information. I found myself a little disappointed in the height of the screen; 40 bits isn't tall enough to display very detailed graphics. The T6963 is capable of handling much larger displays than 240x40 and I'd like to find a larger display to work with.

References

[1] Writing Software for T6963C based Graphic LCDs. 18 April 1998. Steve Lawther. http://ourworld.compuserve.com/homepages/steve_lawther/t6963c.htm

[2] T6963C Dot Matrix Controller Data Sheet. 18 June 2003. Toshiba Corporation.

[3] Linux Kernel Driver For Toshiba T6963C Controller Based Liquid Crystal Displays. 10 Mar 2002. Alexander Frink. http://wwwthep.physik.uni-mainz.de/ frink/lcd-t6963c-0.1.1/README.html

[4] The .bmp file format 1998. Stefan Hetzl. http://www.fortunecity.com/skyscraper/windows/364/bmpffrmt.html.

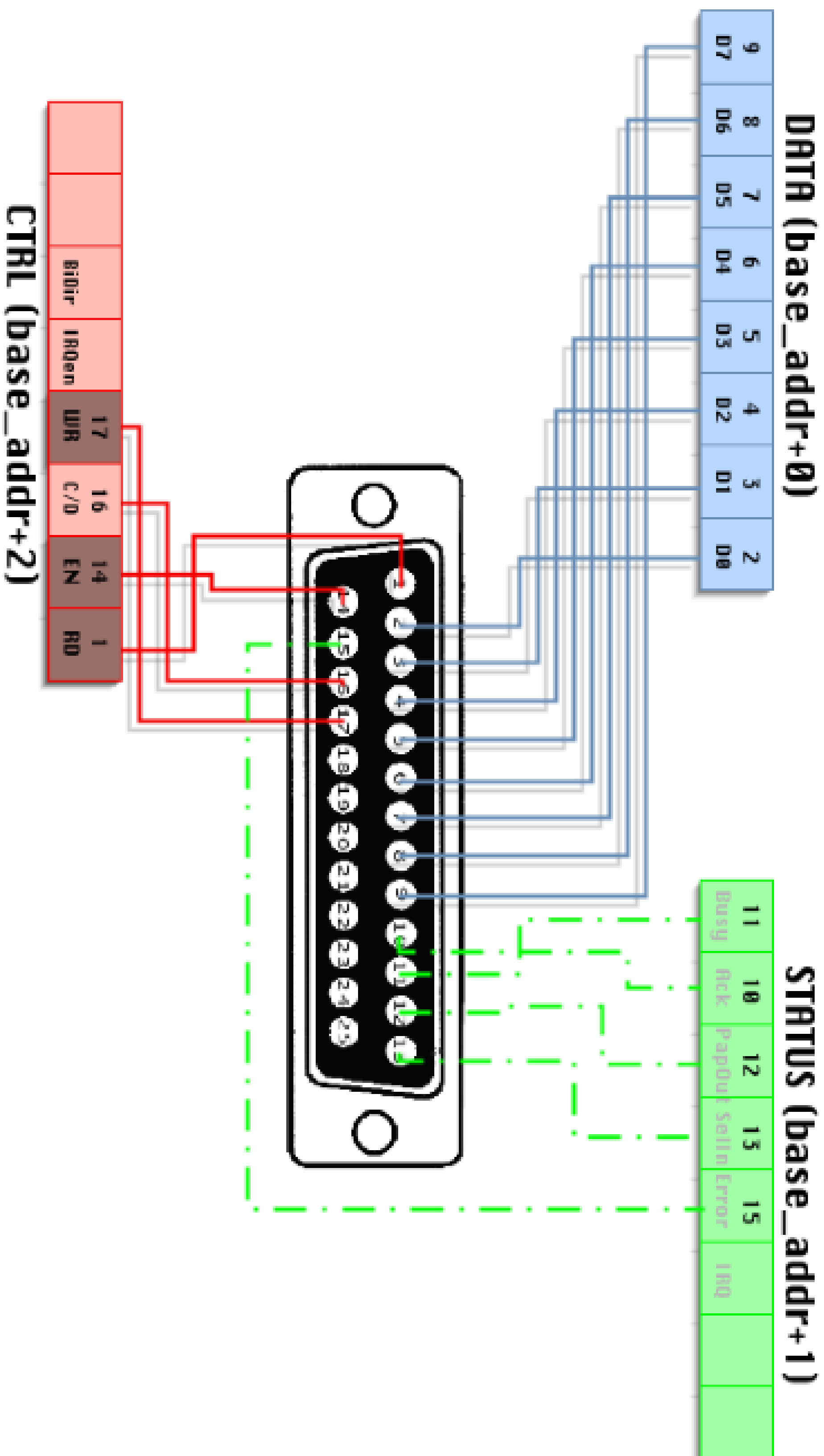

Figure 4: T6963 Parallel Port Connections